Obfuscating Function Call Topography to Test Structural Malware Detection against Evasion Attacks

Andrew Choliy
School of Arts and Sciences
Rutgers University
New Brunswick, New Jersey, U.S.A.
Email: ayc41@scarletmail.rutgers.edu

Feng Li and Tianchong Gao
Department of Computer and Information Technology
Indiana University-Purdue University Indianapolis
Indianapolis, Indiana, U.S.A.
Email: (fengli, tgao)@iupui.edu

***Abstract*—The incredible popularity of the Android mobile operating system has resulted in a massive influx of malicious applications for the platform. This malware can come from a number of sources as Google allows the installation of Android App Packages (APKs) from third parties. Even within its own Google Play storefront, however, malicious software can be found.**

One type of approach to identify malware focuses on the structural properties of the function call graphs (FCGs) extracted from APKs. The aim of this research work is to test the robustness of one example method in this category, named the ACTS (App topologiCal signature through graphleT Sampling) method. By extracting graphlet statistics from a FCG, the ACTS approach is able to efficiently differentiate between benign app samples and malware with good accuracy.

In this work, we obfuscate the FCG of malware in several ways, and test the ACTs method against these evasion attacks. The statistical results of running ACTS against unmodified real malware samples is compared with the results of ACTS running against obfuscated versions of those same apps.

I. Introduction

Android is and likely will continue to be a remarkably successful mobile platform. In fact, it powers more than just smartphones and tablets - in recent years, several set-top boxes and wearable devices have been released with the operating system. However, as Android becomes more and more ubiquitous, it also becomes a more favorable target for malware developers.

Several methods have been proposed to detect and/or defend against this malware, each with their own benefits and drawbacks. App topoligiCal signature through graphleT Sampling (ACTS) is one such approach, and is the focus of this paper. It examines certain structural features of an APK's FCG, so chosen for its immunity to lower-level instruction obfuscation and higher-level content encryption, among other things [1]. Namely, ACTS uses the frequency distribution of various graphlets in a function call graph. A graphlet is small connected subgraph; in ACTS, they consist of no fewer than 3 nodes and no more than 5. The method takes advantage of the fact the much of the malware on Android can be grouped into families which often share some source code and have some degree of similarities in their structure [2].

There are some potential challenges to ACTS, however. While a FCG may be immune to certain aforementioned obfuscation techniques, it may be particularly vulnerable to useless functions and function calls; that is, functions and function calls whose sole purpose is to manipulate the FCG obtained from an APK file. A malicious developer could strategically create functions and/or function calls to manipulate the FCG of their software to resemble a legitimate app, which in turn will fool ACTS. Creating graphlets made up of the useless functions will change their distribution statistics, which is exactly the information that ACTS uses for detection. We experiment with a few potential strategies a malware developer may utilize to evade detection by ACTS. As source code for Android malware is not readily available, we instead directly modify the function call graphs extracted from the malicious APK files via Androguard. While less convenient, it is still reasonably feasible for a malware developer to appropriately add functions and function calls in the source code of their application. Note that any modification we make to the FCG will involve adding a function or function call, as removing those may impair intended operation of the software.

The obfuscation strategies we test ACTS against include random edge (i.e. function call) insertion, creating graphlets via specific edge insertion, and creating graphlets via specific edge and node (i.e. function call and function) insertion. We examine two slight variations of the latter two techniques that aim to reduce the amount of new edges and/or nodes required to successfully obfuscate the FCG. We also briefly discuss ways that

This is the author's manuscript of the article published in final edited form as:
Choliy, A., Li, F., & Gao, T. (2017). Obfuscating function call topography to test structural malware detection against evasion attacks. In 2017 International Conference on Computing, Networking and Communications (ICNC) (pp. 808–813).
https://doi.org/10.1109/ICCNC.2017.7876235

a malicious developer could obfuscate his program that we cannot adequately model by only modifying a FCG.

To summarize, the main contributions of this paper are as follows:

- We identify and design various obfuscation techniques a malicious Android application developer can utilize to evade function call graph based malware detection methods.
- We implement and test these techniques using real malware against the ACTS program and present our results.

The rest of the paper is organized in the following way: In section II, we provide the reader with an overview of the ACTS scheme and give some necessary background information. Section III details the various obfuscation techniques we used in an attempt to evade detection by ACTS. Section IV presents the results of our experiment. The final three sections present related work, potential future research, and concluding remarks, respectively.

II. ACTS Overview

Now we will explore how ACTS works with a little more detail. This section provides enough information to understand how our evasion techniques are implemented, but for a more comprehensive explanation of the program, see [1]. The notation and visual representation we use for graphlets, nodes, etc. are the same as they appear in that paper. The steps ACTS takes to classify malicious APK files is as follows:

A. Extracting the Function Call Graph

First, a function call graph is extracted from an APK file. This FCG is directed graph that represents function caller and callee relationships in a piece of software. Androguard is used to extract a FCG from the Java bytecode of an APK file.

B. Estimation of Graphlet Frequency Distribution (GFD)

Next, ACTS considers the graphlets in the FCG. As previously mentioned, a graphlet is a small, connected subgraph of a FCG. These graphlets capture topological information on local neighborhoods. The set of unique graphlets with 3 nodes may be enumerated as $\omega_{3,i}$ (i = 1, 2, ... 13). Graphical representation of this set is shown in Fig. 1. There are 199 unique 4-node graphlets, 9,364 5-node graphlets, and 1,530,843 6-node graphlets. ACTS only considers graphlets up to size 5, as any more than that will significantly increase computational costs while providing little extra accuracy.

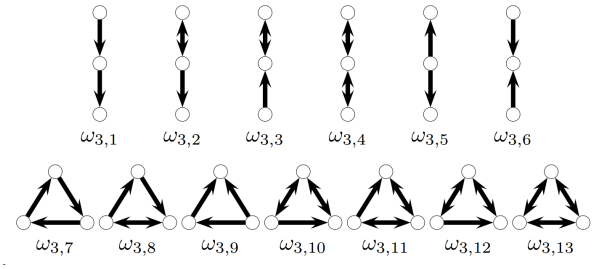

Fig. 1. The thirteen unique graphlets with 3 nodes $\omega_{3,i}(i = 1, 2, ...13)$

ACTS uses the number of occurrences for each type of graphlet to find the Graphlet Frequency Distribution (GFD) of a Graph G. GFD is a vector of the probability distribution of each graphlet type in G. To elaborate, suppose that we enumerate every 3-node graphlet in G (which is finite). Then, define $f_{3,i}$ to be the number of times that graphlet $\omega_{3,i}$ appears in G. Then, the frequency distribution density $d_{3,i} = f_{3,i}/\Sigma_{i=1}^{13} f_{3,i}$. The vector $(d_{3,1}, d_{3,2}, ...d_{3,13})$ is the 3-graphlet frequency distribution (3-GFD) of G. Concatenate this with the 4-GFD and 5-GFD (which will have 199 elements and 9,364 elements, respectively) to get the full 9,576-dimensional Graphlet Frequency Distribution vector of G.

Unfortunately, actually enumerating every graphlet in a FCG is not feasible in a reasonable time frame. It is possible to estimate graphlet distributions without doing so, however. To accomplish this, ACTS makes use of the Metropolis-Hastings algorithm (M-H algorithm) [3], which is a Markov chain Monte Carlo (MCMC) method, to estimate the GFD of graph G. For further details on the estimation method, see [3] or the appropriate section of [1] for more information.

C. Projecting GFD Vector to Lower Dimension

In practice, ACTS does not use the entire GFD vector. Instead, we consider the set of Android applications for which the full GFD vectors have been obtained by Peng et al [1]. From this set, we select graphlets which have a density of 2 percent or more in at least one application. As a result, the number of dimensions of the GFD vector is reduced from 9,576 to 96 (5 3-node graphlets, 20 4-node graphlets, and 71 5-node graphlets). This step is necessary to avoid issues that many machine learning algorithms have when dealing with vectors of high dimensionality [4]. The new GFD vector is referred to as the topological signature of an app.

D. Classification via Support Vector Machine

Lastly, ACTS uses the reduced GFD with the LIBSVM support vector machine (SVM) library to classify its corresponding APK file as malicious or not, with results of up to 87.9 percent accuracy.

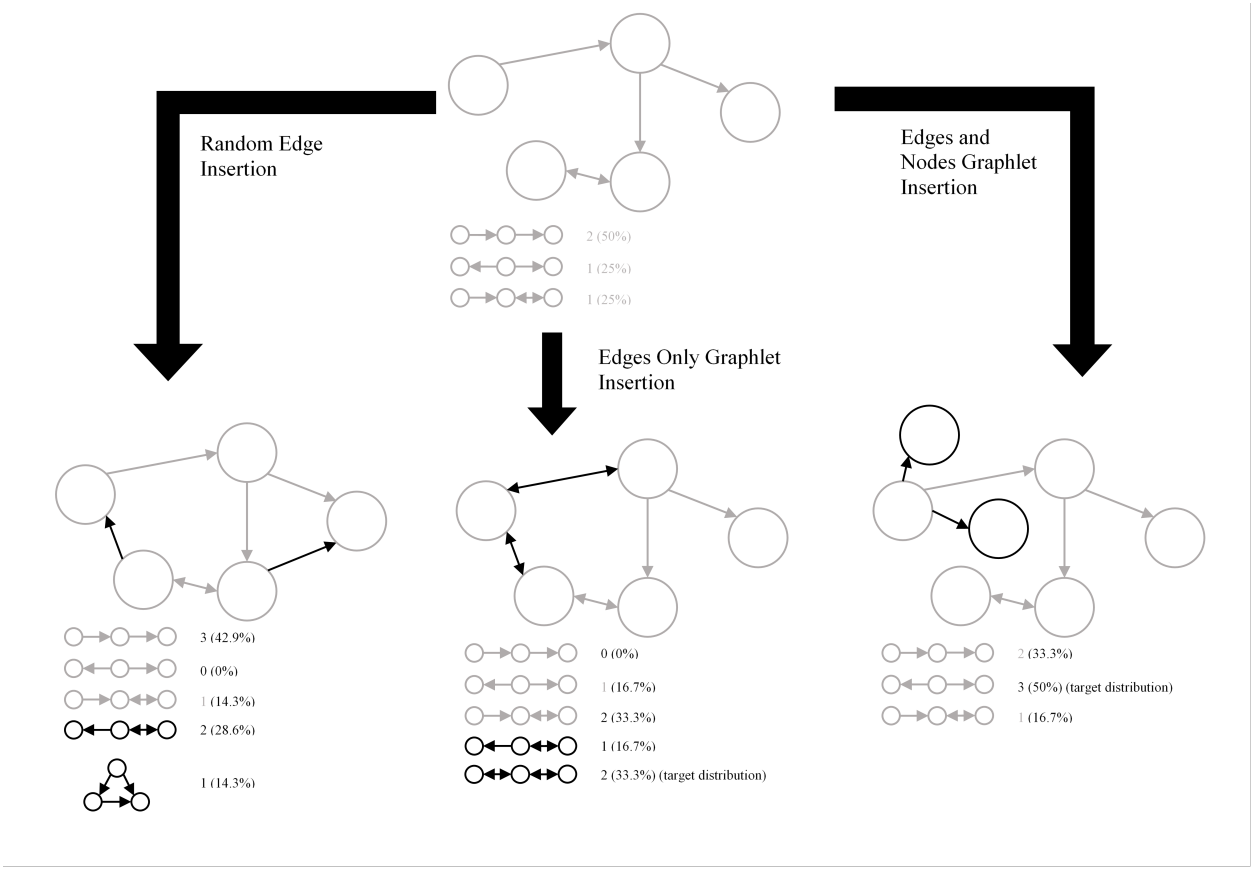

Fig. 2. From left to right, the three obfuscation methods are: Random Edge Insertion (REI), Edges Only Graphlet Insertion (EOGI), and Edges and Nodes Graphlet Insertion (ENGI). In REI, we see two new edges randomly added to the FCG. In EOGI, we see three edges added in such a way to increase the distribution of $\omega_{3,4}$ graphlets, and in ENGI, we edges and nodes added in such a way to increase the distribution of $\omega_{3,5}$ graphlets.

III. Method

Here, we elaborate on the various obfuscation techniques we employed on real malicious software in an attempt to evade detection by ACTS. They can be briefly summarized as follows:

1) Random Edge Insertion: We simply insert random edges (i.e. function calls) on an APK's FCG. This serves as a sort of baseline that shows some degree of robustness for ACTS, as well as how our other strategies perform.
2) Edges Only Graphlet Insertion: We create only edges in the FCG to link existing nodes so that they form graphlets that do not appear often enough in a malicious FCG.
3) Edges and Nodes Graphlet Insertion: We create both edges and nodes to form graphlets that do not appear often enough in a malicious FCG, and form a few more edges to link back to the main body of the graph.

Fig. 2 shows a visual representation of how each presented obfuscation strategy works.

A. Random Edge Insertion

The first technique we employ is Random Edge Insertion (REI). It simply consists of randomly finding two unconnected nodes in a graph G, and adding a directed edge from one to the other. The amount of total edges in G is increased by 50 percent to 100 percent.

This method may seem ultimately inconsequential and a malware developer would likely not consider this strategy in a real-world scenario, but there is some rationale behind our decision to include it here. As previously mentioned, Android malware can generally be classified into one of several families which usually share some code and underlying structure. This commonality between malware is crucial to how ACTS functions. Benign apps, on the other hand, presumably are not nearly as closely related, so perhaps something as simple as adding random edges to a FCG would enable the malware to avoid detection.

At the very least, REI could be a heuristic indication of ACTS' resistance to basic obfuscation and a baseline to measure our other methods against.

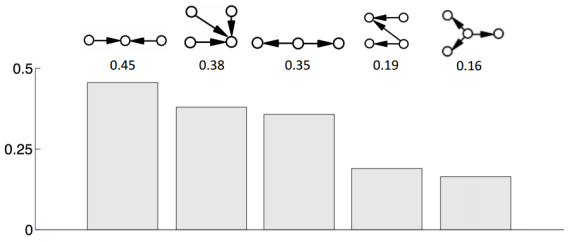

Fig. 3. These graphlets have the highest frequency distributions among the function call graphs of benign Android applications.

B. Edges Only Graphlet Insertion

The second obfuscation strategy we use is Edges Only Graphlet Insertion (EOGI). ACTS stores the GFD of every APK sample it analyzes, as well as statistics about the most common graphlets in malware and benign software by average distribution. Essentially, we treat the statistics of the benign software as a sort of target. We examine the top five graphlets that appear in benign software (see Fig. 3; it is identical to the results obtained by [1]). We then look at the distribution of those graphlets in the malicious FCG, and add edges to existing nodes in order to increase the number of a particular graphlet. These (unconnected) nodes are chosen randomly. So, for example, if the extracted FCG G of some malware has a relative shortage of graphlet $\omega_{3,5}$, we could randomly select three unconnected nodes a, b, and c and add an edge from a to b and from a to c. Each graphlet type will require its own specific manipulation of the FCG.

The above example illustrates the basic concept of EOGI, but we often deal with relatively large FCGs that can have thousands of nodes. In the interest of minimizing edge additions to optimize obfuscation, we consider two slightly more sophisticated versions of this method:

- The first involves adding a substructure that contains multiple instances of a graphlet while lowering the number of edges added per graphlet. Fig. 4 (a) illustrates the following example: consider the previous scenario where a malicious FCG has a shortage of graphlet $\omega_{3,5}$. Now we randomly select 6 nodes d, e, f, g, h, and i. Add an edge from d to every other selected node. As the illustration shows, this structures results in ten $\omega_{3,5}$ graphlets being added even though only five edges were added; much improved over adding three edges for one graphlet. Similar to before, each graphlet will need a specially designed substructure to achieve this effect.
- The second version attempts to find existing subgraphlets in a FCG and add edges to transition it into a specific graphlet. Going back to the scenario where there were too few $\omega_{3,5}$ graphlets, the process is as follows (depicted in Fig. 4 (b)): select a random node j. Ensure that it already has at least one outgoing edge, and then add another edge to an unconnected node. This forms at least one instance of graphlet $\omega_{3,5}$. Once again, different graphlets have different variations of this technique.

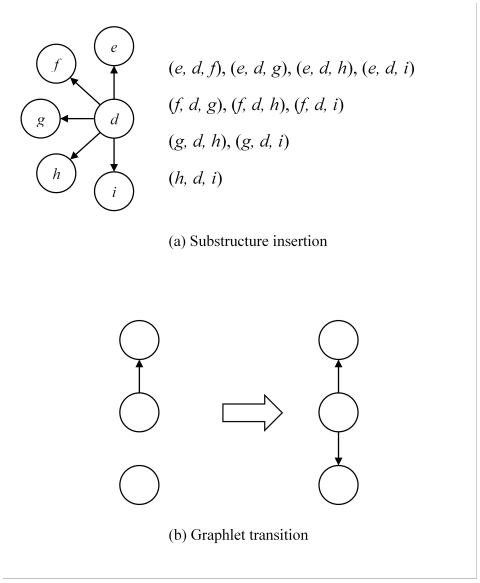

Fig. 4. In (a) inserting that particular substructure increases the amount of $\omega_{3,5}$ graphlets by 10. In (b), we add only one edge to create another instance of graphlet $\omega_{3,5}$.

C. Edges and Nodes Graphlet Insertion

A weakness of EOGI is the potential ripple effect that each added edge can have. Edges are often part of numerous graphlets, and an added edge may inadvertently cause the distribution of undesirable graphlets to increase. Additionally, it is possible to not even add the correct graphlet with EOGI. Consider an attempt to increase the distribution of graphlet $\omega_{3,5}$ by adding an add edge to an existing subgraphlet that consists of two nodes with one edge between them, as shown in Fig. 4 (b). However, if the node that an edge is being made to already has an edge with another node in the graphlet (i.e. if the bottom node is already connected to the top node), it will cause the distribution of graphlet $\omega_{3,9}$ to increase instead.

To mitigate the impact of this phenomenon, we consider a third obfuscation strategy: Edges and Nodes Graphlet Insertion (ENGI). By and large, it follows the same logic as EOGI, but nodes are created instead of selected. The two variations of ENGI are similar to those of EOGI as well. The example presented in Fig. 4 can just as easily be used to illustrate to ENGI. In the case of the substructure insertion variant, ENGI's distinction

is that nodes e, f, g, and h are created. Nodes d and i are ones that already existed in the FCG, and are used to connect the substructure to the main graph. GFD estimation will not consider unconnected sections. When using the graphlet transition variant of ENGI, the middle and upper nodes as well as the upper edge are already present in the FCG, while the bottom node and edge are created.

IV. Results

To test our obfuscation techniques, we considered 20 random FCGs that came from known malware. In ACTS, these were all true positives; that is, correctly classified as malware. We applied each technique on the samples and recorded any difference in ACTS' classification. The results of our experiments are promising. As Fig. 5 shows, both ENGI's and EOGI's substructure insertion variant performed well, both causing ACTS to change its original correct malicious classification of 17 malicious FCGs from a total of 20. EOGI's substructure insertion performance came as a surprise, however. Intuitively, the number of unintended graphlet additions would leave too much of an impact for good results (which we feel may be the case for the graphlet transition variant of EOGI), but the data indicates otherwise. The graphlet transition version of ENGI also performed well with 14 misclassifications, though EOGI's version did poorly with only 6. REI's poor performance suggests that even though much of Android can fit into one of several structural molds, simply disrupting this structure is not enough. Legitimate pieces of software may not by as easy to categorize, but there are still some conventions in the source code that are reflected in a FCG. REI was not able to reflect this in the modified graph.

A. Discussion

An important element of this obfuscation to consider is practicality and cost for the malicious developer. We found that the number of graphlets required to change the GFD to a desirable value was anywhere from a few hundred to several thousand, depending on the size of the FCG and its current GFD. Often, however, this means that the number of edges and nodes (when applicable) increases by several times. While not necessarily difficult, adding all of those functions and function calls to a program can be time-consuming. EOGI is more feasible than ENGI, as only function calls are added. There are several things that can be done to mitigate this issue, however. Our implementation of the substructure strategy involved six nodes and five edges, and resulted in 10 graphlets being added. Adding another node and edge to it would cause 15 graphlets to be added; yet another node and edge would further increase that to 21, and so on. A large enough substructure will reduce

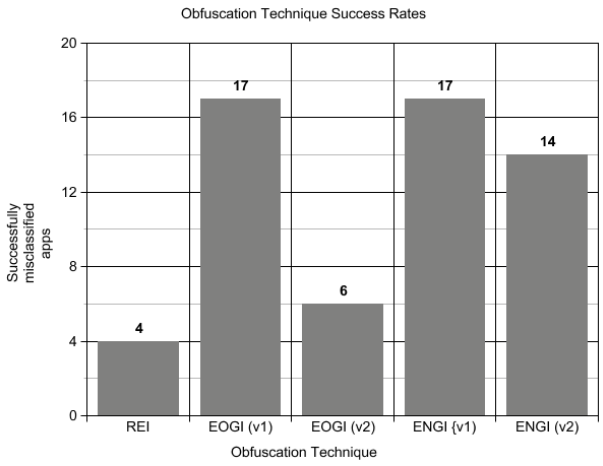

Fig. 5. This figure shows how many apps were misclassified by ACTS after our obfuscation techniques were applied to them. Here, v1 is the substructure insertion variant of EOGI/ENGI, and v2 is the graphlet transition variant.

the workload. Adopting a similar idea with the graphlet transition strategy would make it virtually indistinguishable from structure insertion; the only difference is that it would guarantee the presence of a subgraphlet that existed in the original FCG. Perhaps a more advanced version of graphlet transition could identify multiple subgraphlets or graphlets and connect them together to create a desired substructure. The process of finding suitable subgraphlets and graphlets may be non-trivial, however. Additionally, the malicious developer is able to remove or condense functions, which means he or she can modify a FCG in new ways.

V. Related Work

There is little to no research that focuses specifically on obfuscating the function call graphs of applications as we have done here. The idea itself of using a FCG to detect malicious software is relatively novel. Obviously, Peng et al. [1] created ACTS. [5] implements a similar approach, though they also take into account features of each node in the graph, not just the overall structure. Testing our obfuscation techniques and devising new ones to combat their detection could be interesting. FCG analysis can also be used for other purposes; for instance, Zhou and Jiang [6] use it to detect specific vulnerabilities in Android applications.

VI. Future Work

Aside from further investigation and refinement of the presented obfuscation techniques, there are two main areas we believe will result in the most interesting and practical related research. The first is studying other FCG obfuscation techniques, especially ones that require direct editing of the source code (and thus are impossible

to do with our own approach). One such example would be function inlining, which essentially involves condensing several function into one larger function. This could enable the deletion of nodes and edges in the FCG, which our approach was unable to do.

The second interesting area of future study is obfuscation detection. Certain bytecode or structural clues could potentially indicate FCG obfuscation. For instance, the nodes (i.e. functions) added by ENGI would generally exist only for other functions to call or be called by. Finding these "useless" functions even with just bytecode is possible, and indeed could be gleaned by using [5]'s FCG node and neighborhood analysis. [5] also found that the average number of outgoing function calls from a node is slightly above two, so detecting REI or EOGI could be made possible by using that and perhaps other statistics.

VII. Conclusion

In this paper, we present practical obfuscation techniques that malicious Android software developers could implement in order to avoid FCG-based malware detection. Namely, we show the effectiveness of some of these techniques against an example of this detection software: ACTS. By adding edges or nodes and edges, we are able to manipulate a malicious FCG to sufficiently change its graphlet frequency distribution vector, thus fooling ACTS into classifying it as benign software.

References

[1] W. Peng, T. Gao, D. Sisodia, T. K. Saha, F. Li, and M. A. Hasan, ACTS: Extracting Android App Topological Signature through Graphlet Sampling.

[2] Y. Zhou and X. Jiang, Dissecting Android malware: Characterization and evolution. *In Proc. of IEEE Symposium on Security and Privacy (S&P)*, 2012

[3] N. Metropolis, A. W.Rosenbluth, M. N.Rosenbluth, A. H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics Physics*, 21(6): 1087 - 1092, 1953.

[4] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proc of ACM Symposium on Theory of Computing (STOC)*, 1998

[5] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck. Structural detection of Android malware using embedded call graphs. In *Proc. of ACM Workshop on Artificial Intelligence and Security (AISec)*, 2013

[6] Y. Zhou and X. Jiang Detecting Passive Content Leaks and Pollution in Android Applications. In *Proc. of NDSS Symposium*, 2013